Josh Sheldon (Defaultio)

[redacted]

DMCA Takedown Request

Infringed material: Lumber Tycoon 2: www.roblox.com/games/13822889/Lumber-Tycoon-2
Infringing material: Woodmill Inc: www.roblox.com/games/6995451513/Woodmill-Inc

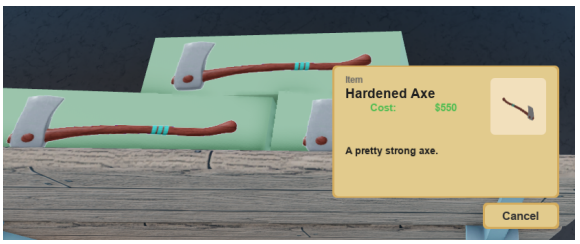
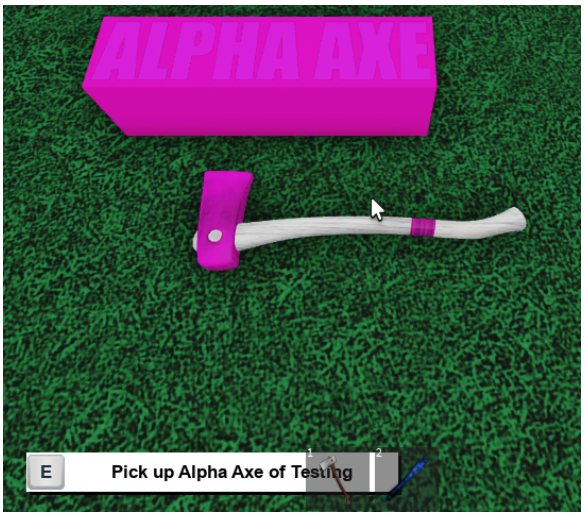
Summary

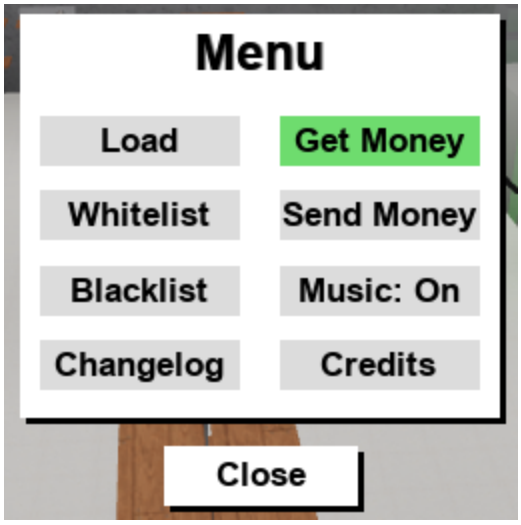
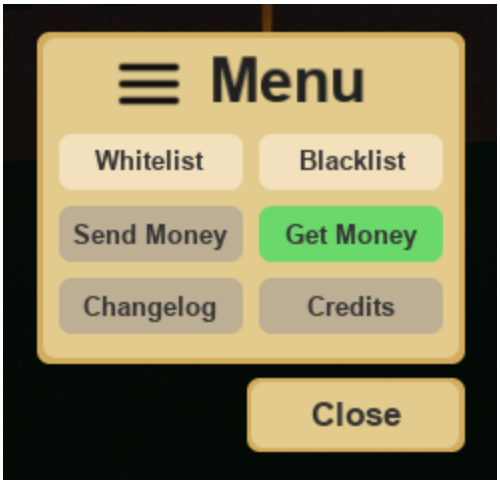
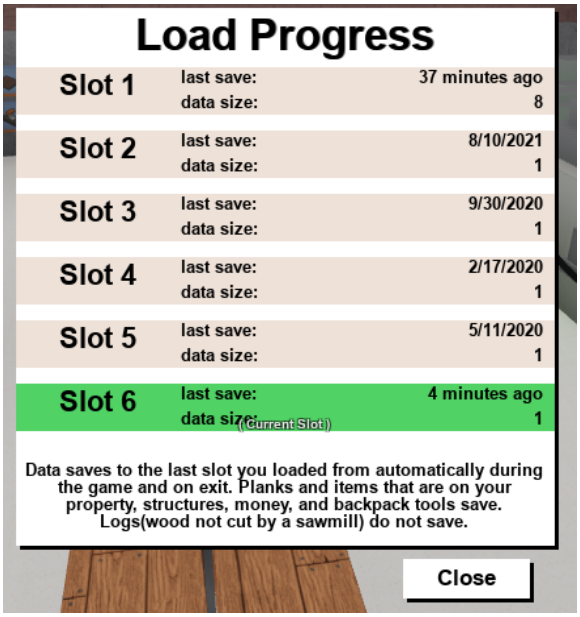
The developers of Woodmill Inc appear to intend to create and monetize a clone of the game mechanics and content of Lumber Tycoon 2. All of the game mechanics are identical, most GUIs are identical in layout and functionality, ingame character dialogue is very similar, many items are similar or identical, and the game takes several assets from the original game. Most offensively, the game appears to be derived heavily from an illicit leaked version of Lumber Tycoon 2 from a 2017 Roblox server data breach. As compensation for this data breach, Roblox committed to take down reuploads and handle damages in perpetuity.

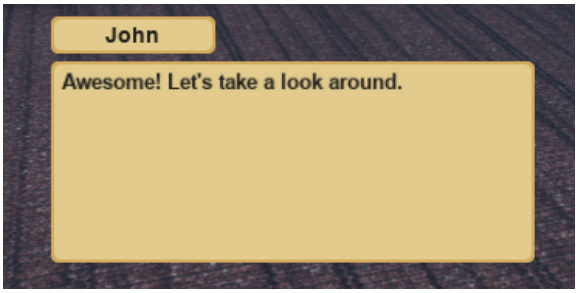
In-Game Similarities

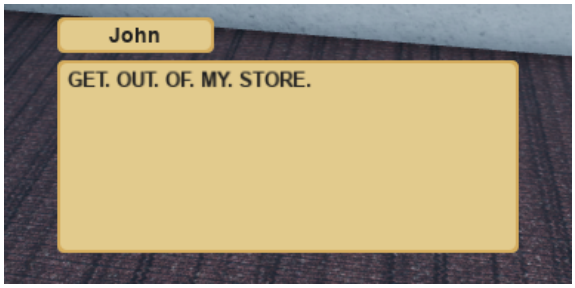
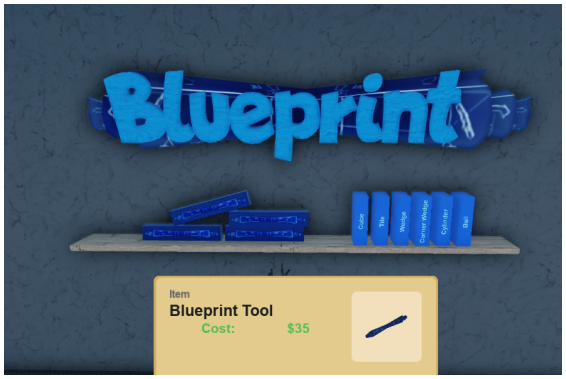
The following is a set of comparison screenshots between Lumber Tycoon 2 and Woodmill Inc which demonstrate some of the similarities between the games.

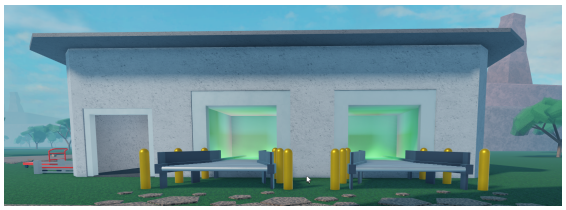
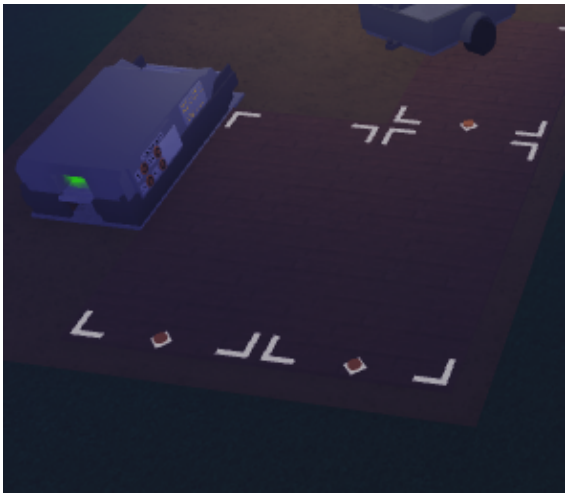
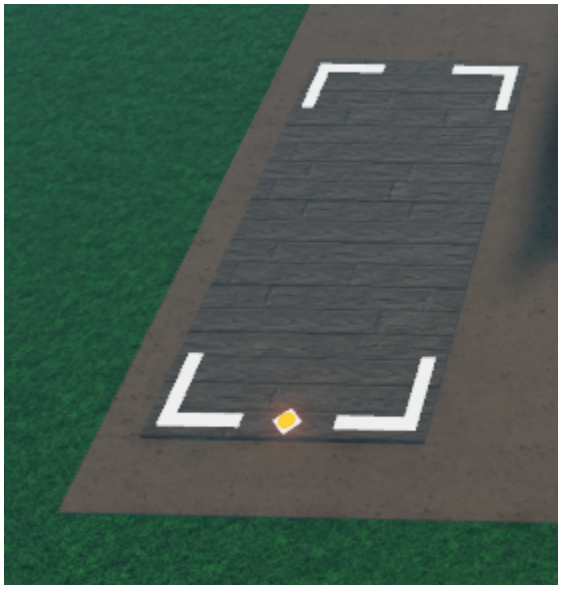
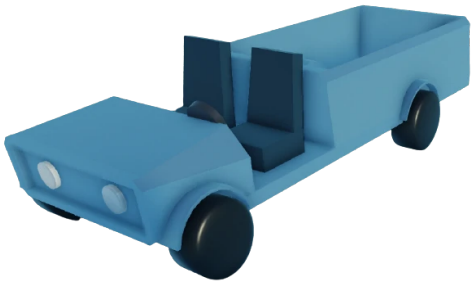
Description	Lumber Tycoon 2	Woodmill Inc
Steel Axe: - Name - Cost - Visual design		

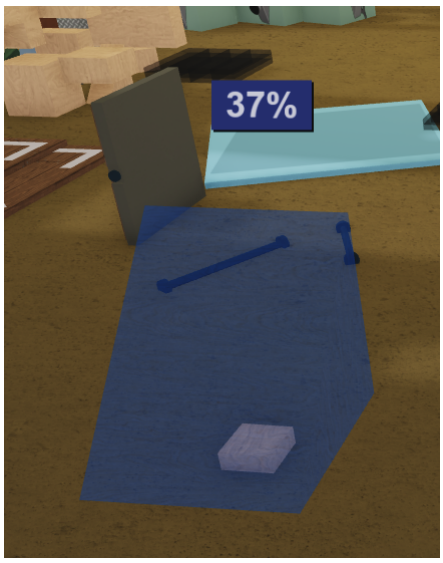
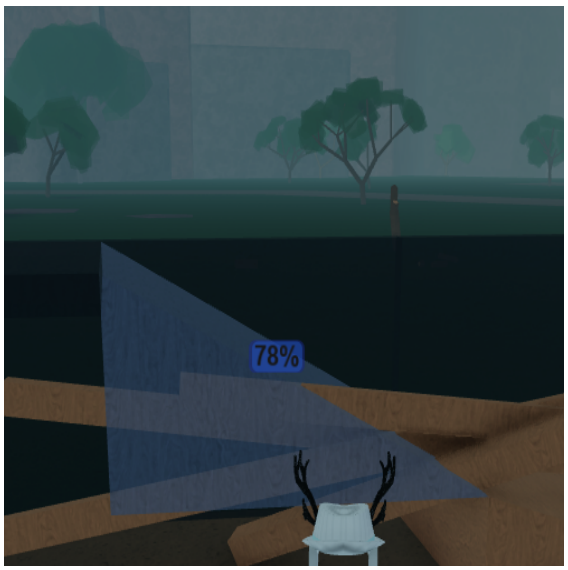
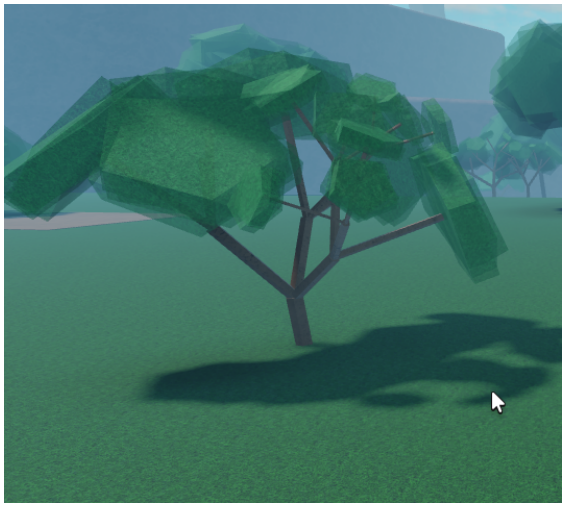
Hardened Axe: - Name - Cost - Visual design		
Plain Axe: - Name - Cost - Visual design		
Basic Hatchet: - Name - Cost - Visual design		
Alpha Axe of Testing: - Name - Visual design		

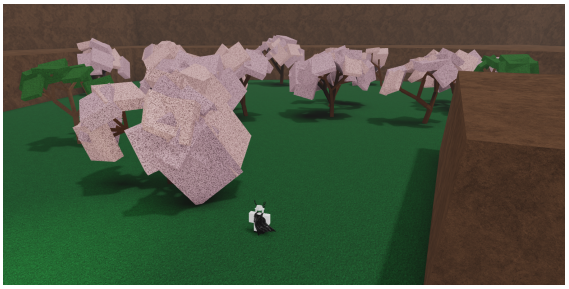
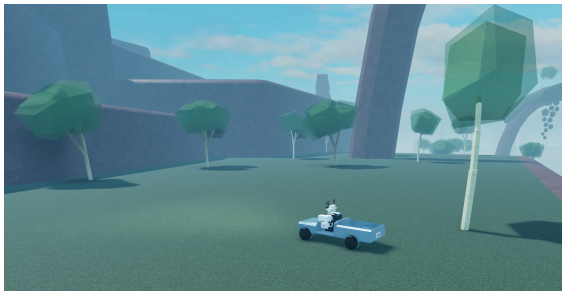
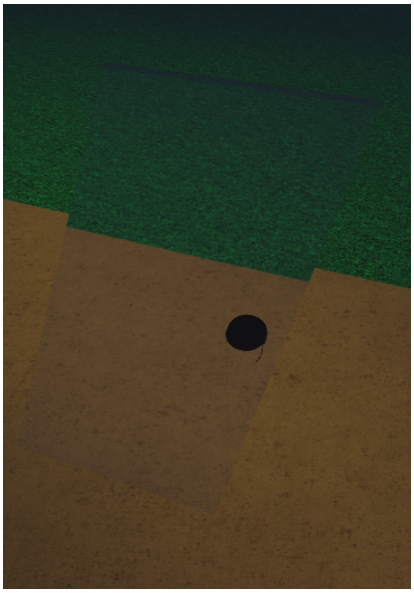
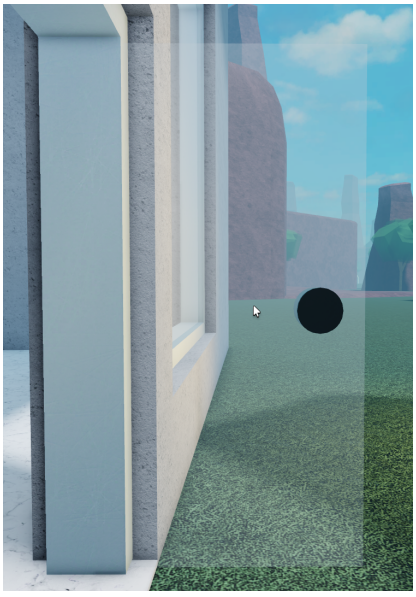
Main menu: - Gui layout and function		
Load screen: - Gui layout and function - Explanation text		
Obscure framerate notice which has not been necessary since 2015		

Land purchasing screen: - Cost of land ($100 + $3000 for each additional plot) - Gui layout and function		
Property sold sign		
Land store owner dialogue when purchasing land		

Land store owner dialogue when user aborts a purchase		
Blueprint section of store		
Tree chopping sound effects	cdn.discordapp.com/attachments/417066380664373258/3349980966963/lt2treechop.mp4	dn.discordapp.com/attachments/417066380664373258/8282703900733/wmitreechop.mp4
Wood milling mechanic	cdn.discordapp.com/attachments/791922763673370627/7585737265172/vDRaeuDS5s.mp4	dn.discordapp.com/attachments/791922763673370627/7584726458398/BFtlyDMbui.mp4
Wood selling mechanic and sound	cdn.discordapp.com/attachments/417066380664373258/2238113783868/2021-09-18_21-37-50.mp4	dn.discordapp.com/attachments/417066380664373258/1170504044564/2021-09-18_21-33-05.mp4

Wood dropoff		
Vehicle spawn pad design and functionality		
Vehicle design		

Blueprint filling mechanic and interface design		
Tree design		
Tree design		

Tree design		
Tree design		
Tree design		
Door design		

Private Assets Used

The following private sound assets from Lumber Tycoon 2 are found in Woodmill Inc without permission:

Asset Name	Use	Link
Chime	Plays when selling wood at wood dropoff	https://www.roblox.com/library/261677648/XylophoneNote
Wood Chop A1	Plays when chopping a tree	https://www.roblox.com/library/260429209/WoodChopA1
Wood Chop A2	Plays when chopping a tree	https://www.roblox.com/library/260429222/WoodChopA2
Wood Chop A3	Plays when chopping a tree	https://www.roblox.com/library/260429234/WoodChopA3
Wood Chop B1	Plays when chopping a tree	https://www.roblox.com/library/260435384/WoodChopB1
Wood Chop B2	Plays when chopping a tree	https://www.roblox.com/library/260435405/WoodChopB2
Wood Chop B3	Plays when chopping a tree	https://www.roblox.com/library/260435420/WoodChopB3
Axe Deflection A1	Plays when an axe is too weak (not implemented in Woodmill Inc, but the asset is still found in the game)	https://www.roblox.com/library/514586161/Hit

Evidence That Woodmill Inc is Based on Lumber Tycoon 2 Source

The developers of Woodmill Inc were determined to prove that their source code is original, so they offered to give me access to their game for inspection. I accepted. I found that the game is indeed not a direct clone of the leaked version of Lumber Tycoon 2, but there is overwhelming evidence that Woodmill Inc is derived from Lumber Tycoon 2.

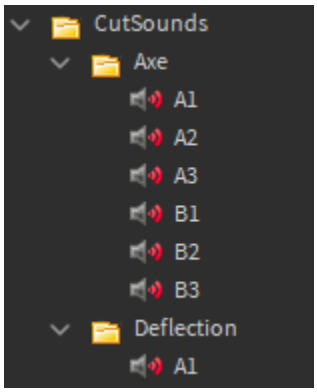
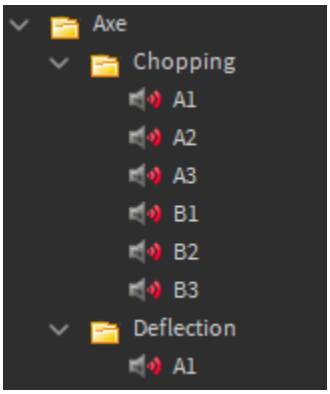
Code Architecture

There are multiple instances in which code content in Woodmill Inc closely mirrors Lumber Tycoon 2. It would be beyond coincidental for them to architect their code so similarly without heavy guidance from the stolen version of Lumber Tycoon 2. The following table demonstrates some of these similarities found in several modules. For brevity, this comparison abbreviates the code by only showing function names and arguments. The content of the full code is also very similar but would be exhausting to include here, so it is included in the *Appendix* section.

Lumber Tycoon 2	Woodmill Inc
ServerScriptService.Trees.TreeUtilities.Cutting function CuttingModule:GetCut(woodClass, player, cutInfo) . . . function CuttingModule:SectionHasChildren(woodClass, sectionId) . . . function CuttingModule:NewCut(sectionId,height) . . . function CuttingModule:UpdateAllCuts(woodClass) . . . function CuttingModule:UpdateCut(woodClass, cut) . . . function CuttingModule:PlayEffects(woodClass, depth, height, sectionId, faceId, hitPoints, cuttingToolClass, cuttingTool) . . . function CuttingModule:DetachBranch(woodClass, cut) . . . function CuttingModule:createNewLooseWood(woodClass, cut, newBranch) . . . function compileUpperSections(woodClass, sectionId, sectionsAboveCut)	*ServerScriptService.Axe.Chopping* function chopping:GetSubBranches(wood_class, branch_id, branches_above_cut) . . . function chopping:CheckCut(wood_class, player, cut_info) . . . function chopping:CreateEffect(wood_class, cut, exact_cframe) . . . function chopping:UpdateCut(wood_class, cut) . . . function chopping:CreateCut(id, height) . . . function chopping:DetachBranch(wood_class, cut) . . . function chopping:CreatePlank(wood_class, branch, side) . . . function chopping:CreateNewLooseWood(wood_class, cut, new_branch)
ServerScriptService.Trees.TreeSuperClass function SuperTree.new(ParentModel, BranchParentTree, BranchParentSection) . . . function SuperTree:GrowCheck(timePassed, acceleratedGrowth) . . . function SuperTree:CanSeedHere(cframe, otherTrees) . . . function SuperTree:PlantSeed(cframe, distUpBranch) . . . function SuperTree:Destroy() . . . function SuperTree:Age(timePassed) . . . function SuperTree:Grow(timePassed) . . . function SuperTree:GetVolume() . . . function SuperTree:ConeCheck(section, nosectioncframe) . . . function SuperTree:Split(parentBranch, type, distUpBranch, numSplits, splitAngle, unitYComponentConstraints, allowableAngleBetweenSplits) . . . function SuperTree:NewSection(cframe, originType, distanceUpParentBranch, parentId, bendAngle) . . . function SuperTree:NewLeafBunch(section) . . . function SuperTree:UpdateLeaves(section, leavesIn) function SuperTree:DestroyLeaves(section) . . . function SuperTree:GetCut(player, ...) . . . function SuperTree:UpdateAllCuts()	*ServerScriptService.Trees.TreeMainClass* function Tree.new(parent : Instance) . . . function Tree:SpawnCheck(location : CFrame, tree_list : table) . . . function Tree:GetTotalVolume() . . . function Tree:RandomFloatFromTable(list : table) . . . function Tree:RandomIntFromTable(list : table) function Tree:CreatePart(part_type : string) . . . function Tree:Plant(location : CFrame) . . . function Tree:CreateBranch(location : CFrame, branch_type : string, parent : number) . . . function Tree:CreateLeaves(branch : table) . . . function Tree:RefreshLeaves(branch : table, leaves : table) function Tree:DestroyLeaves(branch : table) . . . function Tree:SplitBranches(branch : table) function Tree:UpdateDeathTimer(seconds : integer) . . . function Tree:Grow(bypass_growth_limit : boolean) . . . function Tree:CheckForGrowth(growth_time : integer, age_time : integer, increased_growth : boolean)

Sound Assets

The following images are excerpts from the game explorers of Lumber Tycoon 2 and Woodmill Inc. The same assets with the same names and organization can be found in both games. The axe deflection sound is also found in Woodmill Inc, despite it being unused in the game. This suggests that the folders were blindly copied from the Lumber Tycoon 2 leak.

Lumber Tycoon 2	Woodmill Inc
	

Tree Generation Parameters

There are countless possible solutions to parameterizing random tree generation, so any two independently imagined solutions to this problem should be obviously unique. The following compares tree class modules found in each game. Note similarities not only in the names and setup of the parameters, but also their values for these specific tree species, which are all either identical or just slightly modified.

Lumber Tycoon 2	Woodmill Inc
ServerScriptService.Trees.TreeClasses.Generic	*ServerScriptService.Trees.TreeClasses.Generic*

```
SuperTree = require(game.ServerScriptService.Trees.TreeSuperClass)

Tree = {}
Tree.__index = Tree
setmetatable(Tree, SuperTree)

function Tree.new(...)
    local newtree = SuperTree.new(...)
    setmetatable(newtree, Tree)

   --------------[[ Wood Stats ]]--------------

   newtree.Name = script.Name
   newtree.Hardness = 2 --Hit points per cross sectional square stud
   newtree.LogValue = 1.5
   newtree.PlankValue = 10

   --------------[[ Apearance ]]--------------

   newtree.WoodColor = BrickColor.new("Nougat")
   newtree.WoodMaterial = Enum.Material.Wood
   newtree.BarkColor = BrickColor.new("Brown")
   newtree.BarkMaterial = Enum.Material.Concrete
   newtree.BarkThickness = 0.08

   newtree.LeafColors={
      {
         Material=Enum.Material.Grass,
```

```
local TreeMainClass =
require(game.ServerScriptService.Trees.TreeMainClass)

local CustomTree = {}
CustomTree.__index = CustomTree
setmetatable(CustomTree, TreeMainClass)

function CustomTree.new(...)
   local NewTree = TreeMainClass.new(...)
   setmetatable(NewTree, CustomTree)

   --[[ Core Values ]]--

   NewTree.name = script.Name
   NewTree.log_strength = 2.5
   NewTree.plank_strength = 2.5
   NewTree.bark_value = 1.2
   NewTree.plank_value = 11

   --[[ Cosmetic Appearance ]]--

   NewTree.bark_information = {
      material = Enum.Material.Concrete;
      color = BrickColor.new("Pine Cone").Color;
      transparency = 0;
      mesh = game.ServerScriptService.Trees.TreeMeshes.Pentagon;
   }
```

```
        Color=BrickColor.new("Bright green")
    }
}

newtree.BranchClass = nil

--------------[[ Growth Behavior ]]--------------

newtree.MinSpawnDistanceToOtherTrees = 15

newtree.GrowInterval = {  --Seconds between :Grow() is called
    min=3,
    max=4
}

newtree.MaxGrowCalls = {  --Max distance from bottom of trunk to tip
of farthest extremety
    min=40,
    max=55
}

newtree.NewBranchCutoff = 10
newtree.LifetimePerVolume = 45
newtree.LeafDropTime = 80


newtree.SeedThickness = {
    min=0.3,
    max=0.5
}
newtree.ThicknessGrow = { --Amount the outer diameter thickens for
each call of :Grow()
    min=0.015,
    max=0.022
}
newtree.LengthGrow = { --Amount length of extremety branches
increases for each call of :Grow()
    min=0.4,
    max=0.55
}
newtree.BendThicknessReduce = {  --Starting thickness of new bend
segments, subracted from parent branch
    min=0.7,
    max=0.9
}
newtree.SplitThicknessReduce = {  --Starting thickness of new branch
segments, subracted from parent branch
    min=0.28,
    max=0.35
}


newtree.TrunkAnglePhi = { --Angle away fron vertical normal of
baseplate
    min=0,
    max=10
}
newtree.TrunkAngleTheta = {  --Spin
    min=0,
    max=360
}
newtree.TrunkDistanceUntilBending = {  --Will yield these distance
amounts before beginning regular yield cycles for
bending/branching/splitting
    min=10000,
    max=10000
}
newtree.TrunkDistanceUntilSpliting = {
    min=7,
    max=10
}
newtree.TrunkDistanceUntilBranching = {
    min=10,
    max=14
}



newtree.DistanceBetweenSplits = {  --Will yield this distance between
new splits
    min=6,
    max=9
}
newtree.NumSplits = {  --Number of new segments at each split. 1 is
no split.
    min=2,
    max=3
}
newtree.SplitAngle = { --Angle away fron vertical normal of parent
branch
    min=0,
    max=60
}
```

```
NewTree.bark_margin = 0.2

NewTree.plank_information = {
    material = Enum.Material.Wood;
    color = BrickColor.new("Nougat").Color;
    transparency = 0;
}

NewTree.leaf_information = {
    {
        material = Enum.Material.Grass;
        color = BrickColor.new("Bright green").Color;
        transparency = 0.5;
        ignore_relativity = false;
    },
    {
        material = Enum.Material.Grass;
        color = BrickColor.new("Sea green").Color;
        transparency = 0.5;
        ignore_relativity = false;
    }
}

NewTree.leaf_mesh =
game.ServerScriptService.Trees.TreeMeshes.Pentagon

--[[ Growth Requirements ]]--

NewTree.safety_distance = 75

NewTree.growth_limit = {
    minimum = 40;
    maximum = 50;
}

NewTree.growth_rate = {
    minimum = 2;
    maximum = 4;
}

NewTree.lifetime = 60

--[[ Growth Propertries ]]--

-- Initial Data
NewTree.starting_size = {
    minimum = 0.25;
    maximum = 0.5;
}

NewTree.starting_angle = {
    minimum = -16;
    maximum = 16;
}

NewTree.growth_width_size_amount = {
    minimum = 0.01;
    maximum = 0.02;
}

NewTree.growth_height_size_amount = {
    minimum = 0.3;
    maximum = 0.5;
}

-- Branch Data

NewTree.root_branch_distance = {
    minimum = 4;
    maximum = 6;
}

NewTree.branch_size_loss = {
    minimum = 0.3,
    maximum = 0.4
}

NewTree.branch_distance = {
    minimum = 6;
    maximum = 10;
}

NewTree.branch_angle = {
    minimum = 0;
    maximum = 60;
}

NewTree.angle_clamp = {
    minimum = 45;
    maximum = 125;
}

NewTree.branch_split_amount = {
```

```
   newtree.AllowableAngleBetweenSplits = { --Value between 0 and 180
degrees
      min=50,
      max=130
   }
   newtree.SplitUnitYComponentConstraints = {
      min=0.3,
      max=1
   }

   newtree.DistanceBetweenBends = {  --Will yield this distance between
new bends
      min=3,
      max=5 }
   newtree.BendAngle = {  --Angle away fron vertical normal of parent
branch
      min=10,
      max=30
   }
   newtree.BendUnitYComponentConstraints = {
      min=0,
      max=1
   }

   newtree.DistanceBetweenBranching = {  --Will yield this distance
between new splits
      min=4,
      max=10
   }
   newtree.NumBranches = { --Number of new segments at each split. 1 is
no split.
      min=1,
      max=1
   }
   newtree.BranchAngle = { --Angle away fron vertical normal of parent
branch
      min=0,
      max=80
   }
   newtree.AllowableAngleBetweenBranches = {--Value between 0 and 180
degrees
      min=30,
      max=180
   }
   newtree.BranchUnitYComponentConstraints = {
      min=-0.6,
      max=1
   }

   newtree.NumLeafParts = {
      min=1,
      max=1
   }
   newtree.LeafAngle = {
      X = {min=-10, max=10},
      Y = {min=-20, max=20},
      Z = {min=-10, max=10}
   }
   newtree.LeafSizeFactor = {   --Leaf size as a factor of the thickness
of its branch
      X = {min=10, max=14},
      Y = {min=2, max=3},
       Z = {min=10, max=14}
   }

   newtree.SpaceCheckCone = {
      Distance=10,
      Angle=15
   }
   newtree.NumNewSegmentAttempts = 250

   return newtree
end

return Tree
```

```
      minimum = 2;
      maximum = 3;
   }

   -- Leaf Data
   NewTree.leaf_count = {
      minimum = 2;
      maximum = 4;
   }

   NewTree.leaf_size = {
      x = {
         minimum = 10;
         maximum = 15;
      };
      y = {
         minimum = 2;
         maximum = 3;
      };
      z = {
         minimum = 10;
         maximum = 15;
      };
   }

   NewTree.leaf_orientation = {
      x = {
         minimum = -12;
         maximum = 12;
      };
      y = {
         minimum = -17;
         maximum = 17;
      };
      z = {
         minimum = -12;
         maximum = 12;
      };
   }

   return NewTree
end

return CustomTree
```

Axe Parameters

Similar to the tree generation parameters, axe parameters are also similar or identical. See the following unabbreviated modules:

Lumber Tycoon 2	Woodmill Inc

ReplicatedStorage.Purchasables.Tools.AllTools.Axe1.AxeClass (Ingame name: Plain Axe)
 	
`SuperAxe = require(game.ReplicatedStorage.Purchasables.Tools.AxeSuperClass)`
 	
`Axe = {}`
`Axe.__index = Axe`
`setmetatable(Axe, SuperAxe)`
 	
`function Axe.new(...)`
`  local newAxe = SuperAxe.new(...)`
`  setmetatable(newAxe, Axe)`
 	
`  --------------[[ Behavior ]]--------------`
 	
`  newAxe.Damage = 0.55`
`  newAxe.Range = 6`
`  newAxe.SwingCooldown = 0.73`
 	
`  return newAxe`
`end`
 	
`return Axe`
	ReplicatedStorage.Content.Tools.PlainAxe.Class
	
	`local AxeMain = require(game.ReplicatedStorage.Classes.Tools.AxeMainClass)`
	
	`local Axe = {}`
	`Axe.__index = Axe`
	`setmetatable(Axe, AxeMain)`
	
	`function Axe.new(...)`
	`  local newAxe = AxeMain.new(...)`
	`  setmetatable(newAxe, Axe)`
	
	`  newAxe.chopping_power = 0.55`
	`  newAxe.hitting_range = 6`
	`  newAxe.swing_cooldown = 0.73`
	
	`  return newAxe`
	`end`
	
	`return Axe`

ReplicatedStorage.Purchasables.Tools.AllTools.Axe2.AxeClass (Ingame name: Steel Axe)
 	
`SuperAxe = require(game.ReplicatedStorage.Purchasables.Tools.AxeSuperClass)`
 	
`Axe = {}`
`Axe.__index = Axe`
`setmetatable(Axe, SuperAxe)`
 	
`function Axe.new(...)`
`  local newAxe = SuperAxe.new(...)`
`  setmetatable(newAxe, Axe)`
 	
`  --------------[[ Behavior ]]--------------`
 	
`  newAxe.Damage = 0.93`
`  newAxe.Range = 8`
`  newAxe.SwingCooldown = 0.7`
 	
`  return newAxe`
`end`
 	
`return Axe`
	ReplicatedStorage.Content.Tools.SteelAxe.Class
	
	`local AxeMain = require(game.ReplicatedStorage.Classes.Tools.AxeMainClass)`
	
	`local Axe = {}`
	`Axe.__index = Axe`
	`setmetatable(Axe, AxeMain)`
	
	`function Axe.new(...)`
	`  local newAxe = AxeMain.new(...)`
	`  setmetatable(newAxe, Axe)`
	
	`  newAxe.chopping_power = 0.95`
	`  newAxe.hitting_range = 8`
	`  newAxe.swing_cooldown = 0.7`
	
	`  return newAxe`
	`end`
	
	`return Axe`

ReplicatedStorage.Purchasables.Tools.AllTools.Axe3.AxeClass (Ingame name: Hardened Axe)
 	
`SuperAxe = require(game.ReplicatedStorage.Purchasables.Tools.AxeSuperClass)`
 	
`Axe = {}`
`Axe.__index = Axe`
`setmetatable(Axe, SuperAxe)`
 	
`function Axe.new(...)`
`  local newAxe = SuperAxe.new(...)`
`  setmetatable(newAxe, Axe)`
 	
`  --------------[[ Behavior ]]--------------`
 	
`  newAxe.Damage = 1.45`
`  newAxe.Range = 8`
`  newAxe.SwingCooldown = 0.65`
 	
`  return newAxe`
`end`
 	
`return Axe`
	ReplicatedStorage.Content.Tools.HardenedAxe.Class
	
	`local AxeMain = require(game.ReplicatedStorage.Classes.Tools.AxeMainClass)`
	
	`local Axe = {}`
	`Axe.__index = Axe`
	`setmetatable(Axe, AxeMain)`
	
	`function Axe.new(...)`
	`  local newAxe = AxeMain.new(...)`
	`  setmetatable(newAxe, Axe)`
	
	`  newAxe.chopping_power = 1.45`
	`  newAxe.hitting_range = 8`
	`  newAxe.swing_cooldown = 0.65`
	
	`  return newAxe`
	`end`
	
	`return Axe`

Statements

I have a good faith belief that the use of these materials is not authorized by the copyright owner, its agent, or the law.

The above information in this notice is accurate and, under penalty of perjury, I am the copyright or intellectual property owner.

Josh Sheldon

Appendix

More Exhaustive Code Similarities

The following table more exhaustively demonstrates similarities in the modules which were abbreviated in the *Code Architecture* section. Take note of algorithmic symmetries, identical internal constant values, and in several cases nearly wholly identical code.

<table>
<tr><th>Lumber Tycoon 2</th><th>Woodmill Inc</th></tr>
<tr><td>ServerScriptService.Trees.TreeUtilities.Cutting


```
function CuttingModule:NewCut(sectionId,height)
   local newCut = {}
   newCut.Height = height
   newCut.SectionId = sectionId

   newCut.HitPoints = 0

   newCut.Faces = {}
   for i=1, 4 do
      newCut.Faces[i] = {}
      newCut.Faces[i].HitPoints = 0
   end

   return newCut
end

. . .

function CuttingModule:UpdateCut(woodClass, cut)
   if not cut or cut.Finished then
      return
   end

   if not woodClass.Sections[cut.SectionId] then return end

   local section = woodClass.Sections[cut.SectionId]
   local sectionPart = woodClass.Sections[cut.SectionId].Part

   if not sectionPart:IsDescendantOf(workspace) then return end

   local thick = section.Thickness
   local maxHitPoints = thick ^ 2 * woodClass.Hardness
   if cut.HitPoints > maxHitPoints then
      cut.Finished = true
      self:DetachBranch(woodClass, cut)
      return true
   end

   . . . (Omitted iteration and math for creating cut parts on each face; Woodmill Inc removed this feature when they switched to pentagonal shaped tree cross sections instead of square)

   for faceId, face in pairs(cut.Faces) do
      if face.HitPoints > 0 then

         if not face.Part or not face.Part.Parent then
            face.Part = newPart()
            face.Part.Name = "Cut"
            face.Part.CanCollide = false
            face.Part.BrickColor = BrickColor.new("Really black")
            face.Part.Size = Vector3.new(0.2,0.2,0.2) --Vector3.new(0.2,
0.2, 0.2)
            face.PartMesh = Instance.new("BlockMesh", face.Part)
         end

         . . .

         if face.Weld then
            face.Weld:Destroy()
         end

         face.Weld = Instance.new("Weld")
         face.Weld.Part0 = face.Part
         face.Weld.Part1 = woodClass.Sections[cut.SectionId].Part
```

</td><td>ServerScriptService.Axe.Chopping


```
function chopping:CreateCut(id, height)
   local new_cut = {}
   new_cut.height = height
   new_cut.id = id

   new_cut.part = nil
   new_cut.hit_points = 0

   return new_cut
end

. . .

function chopping:UpdateCut(wood_class, cut)
   if (not cut or not wood_class) then
      return false
   end

   if (not wood_class.branches[cut.id]) then
      return false
   end

   local branch = wood_class.branches[cut.id]
   local branch_part = wood_class.branches[cut.id].part

   local thickness = branch.thickness
   local strength
   if (wood_class.model.Name == "Plank" and wood_class.plank_strength)
then
      strength = thickness ^ 2 * wood_class.plank_strength
   else
      strength = thickness ^ 2 * wood_class.log_strength
   end

   self:CreateEffect(wood_class, cut,
wood_class.branches[cut.id].part.CFrame *
CFrame.new(0,-wood_class.branches[cut.id].part.Size.Y/2,0) *
CFrame.new(0,cut.height,0) )

   if (cut.hit_points > strength) then
      self:DetachBranch(wood_class, cut)
      return true
   end

   if (not cut.part) then
      if (wood_class.branches[cut.id].part:IsA("MeshPart")) then
         --print(wood_class)
         if (wood_class.bark_information.mesh) then
            cut.part = createPart(wood_class.bark_information.mesh)
         else
            cut.part = createPart(true)
         end
      else
         cut.part = createPart()
      end

      cut.part.Anchored = wood_class.branches[cut.id].part.Anchored
      cut.part.Name = "Cut"
```

</td></tr>
</table>

```
         face.Weld.C0 = bottomCF * CFrame.new(0, -cut.Height, 0)
         face.Weld.C1 = sectionCF
         face.Weld.Parent = face.Part
      end
   end
end

. . .

function CuttingModule:PlayEffects(woodClass, depth, height, sectionId,
faceId, hitPoints, cuttingToolClass, cuttingTool)

   if cuttingTool and cuttingTool:FindFirstChild("CutEffects") then
      cuttingTool.CutEffects:Fire()
   end

   local effectsPart = Instance.new("Part")
   effectsPart.CustomPhysicalProperties = PhysicalProperties.new(1, 0.3,
0.5)

   local sectionPart = woodClass.Sections[sectionId].Part
   local sectionCF = CFrame.new(0,
-woodClass.Sections[sectionId].Length/2, 0)
   local bottomCF

   local sectionThickness
   local sectionWidth

   . . . (Again omitted some calculation for square tree cuts)

   effectsPart.Size = Vector3.new(sectionWidth - 0.05, 0.2, 0.2)

   effectsPart.CanCollide = false
   effectsPart.Anchored = true
   effectsPart.BrickColor = BrickColor.new("Hot pink")
   effectsPart.Transparency = 1
   effectsPart.CFrame =  sectionPart.CFrame *  sectionCF * bottomCF *
CFrame.new(0, height, 0) * CFrame.Angles(math.pi/2, 0, 0) * CFrame.new(0,
sectionThickness / 2 - 0.125, 0)

   local particles = script.ChipParticles:Clone()
   particles.Parent = effectsPart
   if depth <= woodClass.BarkThickness or depth >= sectionThickness -
woodClass.BarkThickness then
      particles.Color = ColorSequence.new(woodClass.BarkColor.Color,
woodClass.BarkColor.r * particleColorDecay, woodClass.BarkColor.g *
particleColorDecay, woodClass.BarkColor.b * particleColorDecay)
   else
      particles.Color = ColorSequence.new(woodClass.WoodColor.Color,
woodClass.WoodColor.r * particleColorDecay, woodClass.WoodColor.g *
particleColorDecay, woodClass.WoodColor.b * particleColorDecay)
   end

   if hitPoints == 0 then
      cuttingToolClass = "Deflection"
      particles.Rate = 16 * sectionWidth * 2
   else
      particles.Rate = 16 * sectionWidth * hitPoints
   end

   local cutSounds = script.CutSounds[cuttingToolClass or
"Axe"]:GetChildren()
   local sound = cutSounds[math.random(1, #cutSounds)]:clone()
   sound.Pitch = math.random() * 0.25 + 0.8
   sound.Parent = effectsPart
   effectsPart.Parent = workspace.Effects
   sound:Play()

   coroutine.resume(coroutine.create(function()
      wait(0.2)
      particles.Enabled = false

      wait(particles.Lifetime.Max)
      effectsPart:Destroy()
   end))
end

. . .

function CuttingModule:DetachBranch(woodClass, cut)
   if woodClass.Sections[cut.SectionId].lastCutBreak and tick() -
woodClass.Sections[cut.SectionId].lastCutBreak < 0.4 then
      return
   end
   woodClass.Sections[cut.SectionId].lastCutBreak = tick()

   woodClass.Sections[cut.SectionId].IsExtremity = false
   woodClass.Sections[cut.SectionId].IsGrowing = false

   local isRootTree = cut.SectionId == 1 and
woodClass.Sections[cut.SectionId].Part:IsGrounded()

   -- Parse out branch into two sections
      cut.part.CanCollide = false
      cut.part.Massless = true
      cut.part.Material = Enum.Material.SmoothPlastic

      . . .

   --[[if (cut.part:FindFirstChild("WeldConstraint")) then
      cut.part:FindFirstChild("WeldConstraint"):Destroy()
   end]]

   if (not cut.part.Anchored and not cut.weld) then
      cut.weld = Instance.new("WeldConstraint");
      cut.weld.Part0 = cut.part
      cut.weld.Part1 = wood_class.branches[cut.id].part

      cut.weld.Parent = cut.part
   end

   return false
end

. . .

function chopping:CreateEffect(wood_class, cut, exact_cframe)
   local branch = wood_class.branches[cut.id]
   local branch_part = wood_class.branches[cut.id].part

   local thickness = branch.thickness
   local strength
   if (wood_class.model.Name == "Plank" and wood_class.plank_strength)
then
      strength = thickness ^ 2 * wood_class.plank_strength
   else
      strength = thickness ^ 2 * wood_class.log_strength
   end

   local effect_part = cut.part--createPart(true)
   --[[effect_part.Parent = game.Workspace.TemporaryEffects
   effect_part.Transparency = 1
   effect_part.CanCollide = false
   effect_part.CFrame = exact_cframe
   effect_part.Size = Vector3.new(thickness - 0.1, 0.1, thickness -
0.1)[[]]

   local chop_particles = script.ChopEffect:Clone()
   chop_particles.Parent = effect_part
   chop_particles.Color =
ColorSequence.new(wood_class.bark_information.color)
   chop_particles.Rate = 32 * thickness

   local chop_sound =
game.ReplicatedStorage.Sounds.Axe.Chopping:GetChildren()[math.random(1
,#game.ReplicatedStorage.Sounds.Axe.Chopping:GetChildren())]:Clone()
   chop_sound.Pitch = math.random() * 0.25 + 0.8

   chop_sound.Parent = effect_part
   chop_sound:Play()

   coroutine.wrap(function()
      FastWait(0.2)
      chop_particles.Enabled = false

      FastWait(chop_particles.Lifetime.Max)
      chop_particles:Destroy()
      --effect_part:Destroy()
   end)()
end

. . .

function chopping:DetachBranch(wood_class, cut)
   wood_class.branches[cut.id].growth_enabled = false
   for id,_ in pairs(wood_class.branches) do
      wood_class.branches[id].growth_enabled = false
   end

   local bottom_length = cut.height
   print(wood_class.branches[cut.id])
   local top_length = wood_class.branches[cut.id].height - cut.height

   local bottom_start_cframe = wood_class.branches[cut.id].part.CFrame
* CFrame.new(0, -wood_class.branches[cut.id].height/2, 0)
   local top_start_cframe = bottom_start_cframe * CFrame.new(0,
cut.height, 0)

   for _,part in
pairs(wood_class.branches[cut.id].part:GetDescendants()) do
      for _,weld in
pairs(wood_class.branches[cut.id].part:GetJoints()) do
         weld.Enabled = false
      end
      if (part:IsA("Weld") or part:IsA("WeldConstraint")) then
         part.Enabled = false
```

```
   local bottomLength = cut.Height
   local topLength = woodClass.Sections[cut.SectionId].Length -
cut.Height
   local bottomStartCFrame =
woodClass.Sections[cut.SectionId].Part.CFrame * CFrame.new(0,
-woodClass.Sections[cut.SectionId].Length/2, 0)
   local topStartCFrame = bottomStartCFrame * CFrame.new(0, cut.Height,
0)

   woodClass.Sections[cut.SectionId].Part:BreakJoints()

   woodClass.Sections[cut.SectionId].Length = bottomLength
   woodClass.Sections[cut.SectionId].Part.Size =
Vector3.new(woodClass.Sections[cut.SectionId].Part.Size.X, cut.Height,
woodClass.Sections[cut.SectionId].Part.Size.Z)
   woodClass.Sections[cut.SectionId].Part.CFrame = bottomStartCFrame *
CFrame.new(0, cut.Height/2, 0)


   if topLength < minSliverSize then -- We cut just below a joint, all
child branches will fall
      --print("Close top cut")

InnerWoodModule:WeldInnerWood(woodClass.Sections[cut.SectionId],"BottomIn
ner") --Reweld for the resize

      if woodClass.Sections[cut.SectionId]["TopInner"] then
         woodClass.Sections[cut.SectionId]["TopInner"]:Destroy()
         woodClass.Sections[cut.SectionId]["TopInner"] = nil
      end

InnerWoodModule:CreateInnerWood(woodClass,woodClass.Sections[cut.SectionI
d],"TopInner")

      for   _, id in
pairs(woodClass.Sections[cut.SectionId].ChildrenSections) do
         if woodClass.Sections[id] then

InnerWoodModule:CreateInnerWood(woodClass,woodClass.Sections[id],"BottomI
nner")
            self:createNewLooseWood(woodClass, {SectionId = id})

            woodClass.Sections[id] = nil
         end
      end

   elseif bottomLength < minSliverSize then -- We cut just above a joint,
the clicked branch will fall
      --print("Close bottom cut")

      local newBranch = woodClass.Sections[cut.SectionId]
      newBranch.Part = newBranch.Part
      newBranch.Part.Size = Vector3.new(newBranch.Part.Size.X, topLength,
newBranch.Part.Size.X)
      newBranch.Part.CFrame = topStartCFrame * CFrame.new(0, topLength/2,
0)
      newBranch.Length = topLength

      InnerWoodModule:CreateInnerWood(woodClass,newBranch,"BottomInner")

InnerWoodModule:CreateInnerWood(woodClass,woodClass.Sections[newBranch.Pa
rentId],"TopInner")

      newBranch.ParentId = nil

      self:createNewLooseWood(woodClass, cut, newBranch)

      woodClass.Sections[cut.SectionId] = nil
   else                              -- Regular cut, clicked branch wil
split at cut point
      --print("Regular cut")

      local newBranch = deepcopy(woodClass.Sections[cut.SectionId])
      newBranch.Part = newBranch.Part:clone()
      newBranch.Part.Size = Vector3.new(newBranch.Part.Size.X, topLength,
newBranch.Part.Size.Z)
      newBranch.Part.CFrame = topStartCFrame * CFrame.new(0, topLength/2,
0)
      newBranch.ParentId = nil
      newBranch.Length = topLength

      if not (woodClass.Model.Name == "Plank") then
         if newBranch["TopInner"] then --Reweld for reszie
            newBranch["TopInner"]:Destroy()
            newBranch["TopInner"] = nil
            woodClass.Sections[cut.SectionId]["TopInner"] = nil

InnerWoodModule:CreateInnerWood(woodClass,newBranch,"TopInner")
         end
         if woodClass.Sections[cut.SectionId]["BottomInner"] then
            woodClass.Sections[cut.SectionId]["BottomInner"]:Destroy()
            woodClass.Sections[cut.SectionId]["BottomInner"] = nil
            newBranch["BottomInner"] = nil
      end
   end

   wood_class.branches[cut.id].height = bottom_length
   wood_class.branches[cut.id].part.Size =
Vector3.new(wood_class.branches[cut.id].part.Size.X, cut.height,
wood_class.branches[cut.id].part.Size.Z)

   if (wood_class.branches[cut.id].part.Parent and
wood_class.branches[cut.id].part.Parent.Name == "Plank" and
wood_class.branches[cut.id].part.Parent:FindFirstChild("Configuration"
)) then

wood_class.branches[cut.id].part.Parent:FindFirstChild("Configuration"
):FindFirstChild("PlankSize").Value =
wood_class.branches[cut.id].part.Size
   end

   wood_class.branches[cut.id].part.CFrame = bottom_start_cframe *
CFrame.new(0, cut.height/2, 0)

   if (top_length < 0.3 and wood_class.model.Name ~= "Plank") then
      self:CreatePlank(wood_class, wood_class.branches[cut.id], "Top")

      for _, id in pairs(wood_class.branches[cut.id].children) do
         if (wood_class.branches[id]) then
            self:CreatePlank(wood_class, wood_class.branches[id],
"Bottom")
            self:CreateNewLooseWood(wood_class, {id = id})
            wood_class.branches[id] = nil
         end
      end
   elseif (bottom_length < 0.3 and wood_class.model.Name ~= "Plank")
then
      local new_branch = wood_class.branches[cut.id]
      new_branch.part = new_branch.part
      new_branch.part.Size = Vector3.new(new_branch.part.Size.X,
top_length, new_branch.part.Size.Z)
      new_branch.part.CFrame = top_start_cframe * CFrame.new(0,
top_length/2, 0)
      new_branch.parent = nil
      new_branch.height = top_length

      self:CreatePlank(wood_class, new_branch, "Bottom")
      self:CreateNewLooseWood(wood_class, cut, new_branch)

      wood_class.branches[cut.id] = nil
   else
      if (top_length < 0.05 or bottom_length < 0.05) then
         return
      end

      local new_branch = wood_class.branches[cut.id]
      new_branch.part = new_branch.part:Clone()
      new_branch.leaves = wood_class.branches[cut.id].leaves
      wood_class.branches[cut.id].leaves = {}
      new_branch.part.Size = Vector3.new(new_branch.part.Size.X,
top_length, new_branch.part.Size.Z)
      new_branch.part.CFrame = top_start_cframe * CFrame.new(0,
top_length/2, 0)
      new_branch.parent = nil
      new_branch.height = top_length

      if not (wood_class.branches[cut.id].part and
wood_class.branches[cut.id].part.Parent and
wood_class.branches[cut.id].part.Parent.Name == "Plank") then
         self:CreatePlank(wood_class, wood_class.branches[cut.id],
"Top")
         self:CreatePlank(wood_class, new_branch, "Bottom")
      end

      self:CreateNewLooseWood(wood_class, cut, new_branch)
      wood_class.branches[cut.id].children = {}
   end

   if (wood_class.branches[cut.id]) then
      for _,part in
pairs(wood_class.branches[cut.id].part:GetDescendants()) do
         if (part:IsA("Weld") or part:IsA("WeldConstraint")) then
            part.Enabled = true
         end
         for _,weld in
pairs(wood_class.branches[cut.id].part:GetJoints()) do
            weld.Enabled = true
         end
      end
   end
end

. . .


function chopping:CreateNewLooseWood(wood_class, cut, new_branch)
```

```
InnerWoodModule:CreateInnerWood(woodClass,woodClass.Sections[cut.SectionI
d],"BottomInner")
         end

InnerWoodModule:CreateInnerWood(woodClass,newBranch,"BottomInner")

InnerWoodModule:CreateInnerWood(woodClass,woodClass.Sections[cut.SectionI
d],"TopInner")
      end

      self:createNewLooseWood(woodClass, cut, newBranch)
   end

   if woodClass.ReWeld then
      woodClass.ReWeld()
   end

   --[[if woodClass.Model:FindFirstChild("Reweld") then -- If was chopped
from an already downed tree
      woodClass.Model.Reweld:Fire()
   end]]

   if isRootTree and not woodClass.Model:FindFirstChild("RootCut") then
      local rootCut = Instance.new("BoolValue")
      rootCut.Name = "RootCut"
      rootCut.Parent = woodClass.Model
      CuttingModule:TreeCutDown(woodClass, cut.Player)
   end
end

. . .

function CuttingModule:createNewLooseWood(woodClass, cut, newBranch)
   -- Create new loose wood object
   local newLooseWood = {}
   newLooseWood.Hardness = woodClass.Hardness
   newLooseWood.Name = woodClass.Name
   newLooseWood.Model = Instance.new("Model", workspace.LogModels)

   if woodClass.Model.Name == "Plank" then
      newLooseWood.Model.Name = "Plank"
      newLooseWood.Model.Parent = workspace.PlayerModels
   else
      newLooseWood.Model.Name = "Loose_"..newLooseWood.Name
   end

   woodClass.Model.TreeClass:clone().Parent = newLooseWood.Model
   woodClass.Owner:clone().Parent = newLooseWood.Model

   newLooseWood.Sections = {}
   newLooseWood.Cuts = {}

   --print("Cut ID: "..cut.SectionId)
   newLooseWood.Sections[cut.SectionId] = newBranch

   dropLeaves(woodClass, cut.SectionId, cut.Height)


   local sectionsAboveCut = {}
   if not newBranch then
      sectionsAboveCut = {cut.SectionId}
   end
   compileUpperSections(woodClass, cut.SectionId, sectionsAboveCut)


   for _, id in pairs(sectionsAboveCut) do --Add loose wood into new
table
      if woodClass.Sections[id] then
         newLooseWood.Sections[id] = woodClass.Sections[id]
         --print("Adding "..id

         woodClass.Sections[id].IsGrowing = false
         woodClass.Sections[id] = nil


         for _, branch in pairs(woodClass.Branches) do --Move branches
above cut section into new model
            if branch.ParentId == id then
               branch.Branch.Model.Parent = newLooseWood.Model
            end
         end
      end
   end

   if newBranch then
      for _, branch in pairs(woodClass.Branches) do --Move branches on
cut section into new model
         if branch.Branch.Model.Parent == woodClass.Model and
branch.ParentId == cut.SectionId then
            local branchHeight =
branch.Branch.Model.DistUpParentTreeSection
            if cut.Height <= branchHeight.Value then
```

```
   local new_loose_wood = {}
   new_loose_wood.hardness = wood_class.hardness
   new_loose_wood.name = wood_class.name
   new_loose_wood.model = Instance.new("Model")
   new_loose_wood.model.Parent = game.Workspace.LooseWood

   local last_interacted = Instance.new("IntValue")
   last_interacted.Name = "LastInteracted"
   last_interacted.Value = tick()
   last_interacted.Parent = new_loose_wood.model

   new_loose_wood.leaf_model = Instance.new("Model")
   new_loose_wood.leaf_model.Name = "Leaves"
   new_loose_wood.leaf_model.Parent = new_loose_wood.model

   if (wood_class.model.Name == "Plank") then
      new_loose_wood.model.Name = "Plank"
      new_loose_wood.model.Parent = game.Workspace.PlayerItems

      local config = Instance.new("Configuration")
      config.Parent = new_loose_wood.model

      local type_val = Instance.new("StringValue")
      type_val.Value = "Plank"
      type_val.Name = "Type"
      type_val.Parent = new_loose_wood.model

      local plank_size = Instance.new("Vector3Value")
      plank_size.Name = "PlankSize"
      plank_size.Value = Vector3.new()
      plank_size.Parent = config

      local tree_class =
wood_class.model:FindFirstChild("Configuration"):FindFirstChild("TreeC
lass"):Clone()
      tree_class.Parent =
new_loose_wood.model:FindFirstChild("Configuration")
   else
      new_loose_wood.model.Name = "Loose" .. new_loose_wood.name
      local tree_class =
wood_class.model:FindFirstChild("TreeClass"):Clone()
      tree_class.Parent = new_loose_wood.model
   end

   new_loose_wood.cuts = {}
   new_loose_wood.branches = {}

   new_loose_wood.branches[cut.id] = new_branch

   local branches_above_cut = {}
   if (not new_branch) then
      branches_above_cut = {cut.id}
   end

   self:GetSubBranches(wood_class, cut.id, branches_above_cut)

   for _,branch_id in pairs(branches_above_cut) do
      if (wood_class.branches[branch_id]) then
         new_loose_wood.branches[branch_id] =
wood_class.branches[branch_id]

         if (new_loose_wood.branches[branch_id].leaves) then
            for _,leaf in
pairs(new_loose_wood.branches[branch_id].leaves) do
               if (leaf.part.Parent ~= nil) then
                  leaf.part.Parent = new_loose_wood.leaf_model

                  coroutine.wrap(function()
                     FastWait(math.random(30,60))

                     if (leaf.part) then
                        leaf.part:Destroy()
                     end
                  end)()
               end
            end
         end
      end
   end

   for i,_ in pairs(wood_class.cuts) do
      if (wood_class.cuts[i]) then
         if (wood_class.cuts[i].part) then
            wood_class.cuts[i].part:Destroy()
         end
         wood_class.cuts[i] = nil
      end
   end

   for id,branch in pairs(new_loose_wood.branches) do
      pcall(function()
         branch.part.Parent = new_loose_wood.model
         if (new_loose_wood.model:FindFirstChild("Configuration") and
```

```
               branchHeight.Value = branchHeight.Value - cut.Height
               branch.Branch.Model.Parent = newLooseWood.Model
            end
         end
      end
   end

   local newBranches = {}
   for _, branch in pairs(woodClass.Branches) do --Move branches above
cut section into new model
      if not (branch.Branch.Model.Parent == newLooseWood.Model) then
         table.insert(newBranches, branch)
      end
   end
   woodClass.Branches = newBranches

   for i, _ in pairs(woodClass.Cuts--[[cutsAboveCut]]) do --Destroy cuts
above the cut
      if woodClass.Cuts[i] then
         for _, face in pairs(woodClass.Cuts[i].Faces) do
            if face.Part then
               face.Part:Destroy()
            end
         end
         woodClass.Cuts[i] = nil
      end
   end


   for id, section in pairs(newLooseWood.Sections) do
      --print("Table id: "..id.."   Atual id: "..section.ID)
      pcall(function() --> Will err if section.Part has been destroyed.
         section.Part.Parent = newLooseWood.Model
         if section["TopInner"] then
            section["TopInner"].Parent = newLooseWood.Model
         end
         if section["BottomInner"] then
            section["BottomInner"].Parent = newLooseWood.Model
         end
      end)
   end


game.ServerScriptService.Trees.LooseWoodManagement.NewLooseWood:Fire(newL
ooseWood.Model)
end

. . .

function compileUpperSections(woodClass, sectionId, sectionsAboveCut)
   if not woodClass.Sections[sectionId] then return end

   for _, id in pairs(woodClass.Sections[sectionId].ChildrenSections) do
      if woodClass.Sections[id] then
         table.insert(sectionsAboveCut,id)
         compileUpperSections(woodClass, id, sectionsAboveCut)
      end
   end
end
```

```
new_loose_wood.model:FindFirstChild("Configuration"):FindFirstChild("P
lankSize")) then

new_loose_wood.model:FindFirstChild("Configuration"):FindFirstChild("P
lankSize").Value += branch.part.Size
         end
      end)
   end

   for _,part in pairs(new_loose_wood.model:GetDescendants()) do
      if (part.Name == "Cut") then
         part:Destroy()
      end
   end

   new_loose_wood.owner = wood_class.model:FindFirstChild("Owner") and
wood_class.model:FindFirstChild("Owner").Value

   local LooseWood =
require(game.ServerScriptService.LooseWood.LooseWoodMainClass)
   local global_loose_wood_list =
require(game.ServerScriptService.LooseWood.GlobalLooseWoodList)
   global_loose_wood_list.wood[new_loose_wood.model] =
LooseWood.new(new_loose_wood)
   global_loose_wood_list.wood[new_loose_wood.model]:Initialize()


   local connection;

   connection = new_loose_wood.model.Changed:Connect(function()
      if (not new_loose_wood.model or not new_loose_wood.model.Parent)
then
         global_loose_wood_list.wood[new_loose_wood.model] = nil
         connection:Disconnect()
      end
   end)
end

. . .

function chopping:GetSubBranches(wood_class, branch_id,
branches_above_cut)
   if not wood_class.branches[branch_id] then return end

   for _, id in pairs(wood_class.branches[branch_id].children) do
      if wood_class.branches[id] then
         table.insert(branches_above_cut,id)
         chopping:GetSubBranches(wood_class, id, branches_above_cut)
      end
   end
end
```

ServerScriptService.Trees.TreeSuperClass

```
function SuperTree.new(ParentModel, BranchParentTree,
BranchParentSection)
   local newtree = {}
   setmetatable(newtree, SuperTree)

   newtree.IsABranch = not not BranchParentTree
   newtree.BranchParentTree = BranchParentTree
   newtree.ParentModel = ParentModel

   newtree.Sections={}
   newtree.Cuts={}
   newtree.Branches = {}

   newtree.GrowCalls = 0
   newtree.FinishedGrowing = false
   newtree.AdultStage = "Growing"

   return newtree
end

. . .

function SuperTree:GrowCheck(timePassed, acceleratedGrowth)
   if self.Destroyed then
      return
   end

   if self.AcceleratedGrowth then
      if self.FinishedGrowing then
```

ServerScriptService.Trees.TreeMainClass

```
function Tree.new(parent : Instance)
   local newTree = setmetatable({}, Tree)

   newTree.parent = parent

   newTree.branches = {}
   newTree.cuts = {}

   newTree.growth_cycle_completed = false

   newTree.growth_amount = 0

   return newTree
end

. . .

function Tree:CheckForGrowth(growth_time : integer, age_time :
integer, increased_growth : boolean)
   self.growth_check -= growth_time

   if ((self.growth_check <= 0 or increased_growth) and not
self.growth_cycle_completed) then
      self:Grow()
```

```
         self.Model.Parent = self.ParentModel
         self.AcceleratedGrowth = false
      end
   end

   self.TimeToNextGrow = self.TimeToNextGrow - timePassed

   if (self.TimeToNextGrow <= 0 or acceleratedGrowth) and not
self.FinishedGrowing then
      self:Grow()
      self.TimeToNextGrow = randomRange(self.GrowInterval) +
self.TimeToNextGrow -- Add negative value
   elseif self.FinishedGrowing and not acceleratedGrowth then
      self:Age(timePassed)
   end
end

. . .

function SuperTree:CanSeedHere(cframe, otherTrees)
   for _, tree in pairs(otherTrees) do
      if tree.SeedCFrame then
         if (tree.SeedCFrame.p - cframe.p).magnitude <
self.MinSpawnDistanceToOtherTrees then
            return false
         end
      end
   end
   return true
end

. . .

function SuperTree:PlantSeed(cframe, distUpBranch)
   if not self.IsABranch then
      cframe = cframe * CFrame.Angles(0,
math.rad(randomRange(self.TrunkAngleTheta)),
math.rad(randomRange(self.TrunkAnglePhi)))
   end

   self.SeedCFrame = cframe

   self.MaxGrowCalls = randomRange(self.MaxGrowCalls)

   self.Model = Instance.new("Model")
   self.LeafModel = Instance.new("Model", self.Model)
   self.LeafModel.Name = "Leaves"

   self.Type = Instance.new("StringValue", self.Model)
   self.Type.Name = "TreeClass"
   self.Type.Value = self.Name



   self.CutEvent = Instance.new("BindableEvent", self.Model)
   self.CutEvent.Name = "CutEvent"
   self.CutConnection = self.CutEvent.Event:connect(function(...)
self:GetCut(...) end)

   self:NewSection(cframe, "Seed", distUpBranch)
   self.TimeToNextGrow = randomRange(self.GrowInterval)

   if not self.AcceleratedGrowth then
      self.Model.Parent = self.ParentModel
   end

   if not self.BranchParentTree then
      self.Owner = Instance.new("ObjectValue", self.Model)
      self.Owner.Name = "Owner"
      local lastInteraction = Instance.new("IntValue", self.Owner)
      lastInteraction.Name = "LastInteraction"


game.ServerScriptService.Trees.LooseWoodManagement.RunTreeOwnershipLoop:F
ire(self)
   else
      self.Owner = self.BranchParentTree.Owner
   end
end

. . .

function SuperTree:Grow(timePassed)
   if self.FinishedGrowing then
      return
   end

   self.GrowCalls = self.GrowCalls + 1

   local foundGrowingSection = false

   local existingSections = #self.Sections
   for ID = 1, existingSections do
```

```
      self.growth_check += self:RandomIntFromTable(self.growth_rate)
   elseif (self.growth_cycle_completed and not increased_growth) then
      self:UpdateDeathTimer(age_time)
   end
end

. . .



function Tree:SpawnCheck(location : CFrame, tree_list : table)
   for _,tree in pairs(tree_list) do
      if (tree.root_location) then
         if ((tree.root_location.p - location.p).magnitude <=
(self.safety_distance or 75)) then
            return false
         end
      end
   end

   return true
end

. . .

function Tree:Plant(location : CFrame)
   self.growth_limit = self:RandomIntFromTable(self.growth_limit)
   self.growth_check = self:RandomIntFromTable(self.growth_rate)

   self.model = Instance.new("Model")
   self.model.Parent = self.parent
   self.model.Name = "Tree"

   local class_value = Instance.new("StringValue")
   class_value.Parent = self.model
   class_value.Value = self.name
   class_value.Name = "TreeClass"

   self.leaf_model = Instance.new("Model")
   self.leaf_model.Parent = self.model
   self.leaf_model.Name = "Leaves"

   self.root_location = location * CFrame.Angles(0, math.rad(360),
math.rad(self:RandomIntFromTable(self.starting_angle)))

   local global_tree_list =
require(game.ServerScriptService.Trees.GlobalTreeList)

   -- import into global tree list
   global_tree_list.trees[self.model] = self

   local connection;

   connection = self.model.Changed:Connect(function()
      if (not self.model:FindFirstChildOfClass("Part")) then
         self.model:Destroy()
      end
      if (not self.model or not self.model.Parent) then
         global_tree_list.trees[self.model] = nil
         connection:Disconnect()
      end
   end)

   self:CreateBranch(self.root_location, "Root", nil)
end

. . .



function Tree:Grow(bypass_growth_limit : boolean)
   -- Bypass growth limit (for commands, event, or something?)
   if (self.growth_cycle_completed and not bypass_growth_limit) then
      return
   end

   -- Lower growth time at night
   if (game.Lighting.Night.Value == true) then
      if (math.random(1,100) > 25) then
         return
      end
```

```
      local section = self.Sections[ID]
      if section and section.IsGrowing then
         foundGrowingSection = true

         local stopForBranches = self.MaxGrowCalls - self.GrowCalls <
self.NewBranchCutoff and self.BranchClass

         if section.IsExtremety and not stopForBranches and
self:ConeCheck(section) then

            local lengthGrow = randomRange(self.LengthGrow)

            section.Length = section.Length + lengthGrow
            section.DistanceToBend = section.DistanceToBend - lengthGrow
            section.DistanceToSplit = section.DistanceToSplit -
lengthGrow
            section.DistanceToBranch = section.DistanceToBranch -
lengthGrow

            if section.DistanceToSplit <= 0 and section.Length > 2 then
               self.Sections[ID].IsExtremety = false
               self:Split(section, "Split", section.Length,
self.NumSplits, self.SplitAngle, self.SplitUnitYComponentConstraints,
self.AllowableAngleBetweenSplits)
            elseif section.DistanceToBend <=0 and section.Length > 2 then
--Time to bend
               self.Sections[ID].IsExtremety = false
               self:Split(section, "Bend", section.Length, {min=1,max=1},
self.BendAngle, self.BendUnitYComponentConstraints, {min=0,max=180})
            elseif section.DistanceToBranch <=0 and self.BranchClass then
--Time to branch
               section.DistanceToBranch =
randomRange(self.DistanceBetweenBranching)
               self:Split(section, "Branch", section.Length,
self.NumBranches, self.BranchAngle, self.BranchUnitYComponentConstraints,
self.AllowableAngleBetweenBranches)
               --TODO after prototype

            end

         end
         section.Thickness = section.Thickness +
randomRange(self.ThicknessGrow)

         section.Part.Size = Vector3.new(section.Thickness,
section.Length, section.Thickness)
         section.Part.CFrame = section.StartCFrame * CFrame.new(0,
section.Length / 2, 0)

         self:UpdateLeaves(section)
      end
   end

   if self.GrowCalls > self.MaxGrowCalls or not foundGrowingSection then
      self.FinishedGrowing = true
      self.FinalVolume = self:GetVolume()
      self.TimeUntilDeath = self.FinalVolume * self.LifetimePerVolume
   end

   self:UpdateAllCuts()

   for i, branch in pairs(self.Branches) do
      branch.Branch:Grow(timePassed)
      if not branch.Branch.Model.Parent then
         self.Branches[i] = nil
      end
   end
end

. . .

function SuperTree:GetVolume()
   local volume = 0
   local existingSections = #self.Sections
   for ID = 1, existingSections do
      local section = self.Sections[ID]
      if section and section.Part then
         volume = volume + section.Part.Size.X * section.Part.Size.Y *
section.Part.Size.Z
      end
   end

   for i, branch in pairs(self.Branches) do
      volume = volume + branch.Branch:GetVolume()
   end

   return volume
end

. . .

function SuperTree:Split(parentBranch, type, distUpBranch, numSplits,
splitAngle, unitYComponentConstraints, allowableAngleBetweenSplits)
```

```
   end

   local is_branches_growing = false

   -- pairs doesn't work because we're editing the values!?
   for i=1,#self.branches do
      if not (self.branches[i].growth_enabled) then
         continue
      end

      is_branches_growing = true

      if (#self.branches[i].children <= 0) then
         local height_growth_amount =
self:RandomFloatFromTable(self.growth_height_size_amount)

         self.branches[i].height += height_growth_amount
         self.branches[i].length_till_branch -= height_growth_amount

         if (self.branches[i].length_till_branch <= 0) then
            self:SplitBranches(self.branches[i])
         end
      end

      self.branches[i].thickness +=
self:RandomFloatFromTable(self.growth_width_size_amount)

      -- Refresh cosmetic look
      self.branches[i].part.Size = Vector3.new(
         self.branches[i].thickness,
         self.branches[i].height,
         self.branches[i].thickness
      )

      self.branches[i].part.CFrame = self.branches[i].root_location *
CFrame.new(0, self.branches[i].height / 2, 0)

      self:RefreshLeaves(self.branches[i])
   end

   self.growth_amount += 1

   -- Growth completed
   if (not is_branches_growing or self.growth_amount >
self.growth_limit) then
      self.growth_cycle_completed = true
      self.death_timer = self.lifetime * self:GetTotalVolume()
   end
end

. . .

. . .

function Tree:GetTotalVolume()
   local result_volume = 0

   for _,branch in pairs(self.branches) do
      if (branch and branch.part) then
         -- haha pentagonal prism volume formula 😳
         result_volume += (1 / 4) * math.sqrt(5 * (5 + 2 *
math.sqrt(5))) * math.pow(branch.part.Size.X * 0.62, 2) *
branch.part.Size.Y
      end
   end

   return result_volume
end

. . .
```

```
   local diff1 = self.MaxGrowCalls - self.GrowCalls
   local diff2 = self.NewBranchCutoff
   if self.BranchParentTree then
      diff2 = self.BranchParentTree.MaxGrowCalls -
self.BranchParentTree.GrowCalls
   end

   if diff1 < self.NewBranchCutoff or diff2 < self.NewBranchCutoff then
      return
   end

   numSplits=randomRangeInteger(numSplits)

   local splitDistanceBounds = {min = math.sqrt(2 - 2 *
math.cos(math.rad(allowableAngleBetweenSplits.min))), max = math.sqrt(2 -
2 * math.cos(math.rad(allowableAngleBetweenSplits.max)))} --Values
between 0 and 2

   local newCFrames = {}
   for i = 1, self.NumNewSegmentAttempts do

      local success = true
      for j = #newCFrames + 1, numSplits do

         local newTheta = randomRange({min = 0, max = math.pi*2})
--http://msemac.redwoods.edu/~darnold/math50c/mathjax/spherical/spherical
2.png
         local newPhi = math.rad(randomRange(splitAngle))
         local angle = CFrame.Angles(0, newTheta, newPhi) *
CFrame.Angles(0, -newTheta, 0)
         local resultantCFrame = parentBranch.StartCFrame * CFrame.new(0,
parentBranch.Length, 0) * angle

         local unitVector = (resultantCFrame * CFrame.Angles(math.pi/2,
0, 0)).lookVector --Check that section attempts do not violate Y unit
vector bounds and do cone check
         if unitVector.Y > unitYComponentConstraints.max or unitVector.Y
< unitYComponentConstraints.min or not self:ConeCheck(nil,
resultantCFrame) then
            success = false
            --print("Y violation")
            break --Throw out new branch, but keep other branches in the
list
         end

         local success2 = true
         for _, cframe in pairs(newCFrames) do --Check that section
attempts do not violate bounds for allowable angle between splits
            local d = ((resultantCFrame * CFrame.new(0,1,0)).p -
(cframe.netCframe * CFrame.new(0,1,0)).p).magnitude
            if d < splitDistanceBounds.min or d > splitDistanceBounds.max
then
               newCFrames = {}
               success2 = false
               --print("Angle violation: "..d)
               break --Throw out new branch, clean out list and start
over
            end
         end


         if not success2 then
            success = false
            break
         end


         table.insert(newCFrames, {netCframe = resultantCFrame, angle =
angle}) --New branch proposal worked; add it to the list
      end

      if success then
         break
      end
   end

   local success = #newCFrames > 0 --Success does not necessarily mean
the amount of desired branches was reached

   if success then
      for _, cframe in pairs(newCFrames) do

self:NewSection(cframe.netCframe,type,distUpBranch,parentBranch.ID,
cframe.angle)
      end
      if not (type == "Branch") then
         self:DestroyLeaves(parentBranch)
      end
   elseif not success and not (type == "Branch") then --Branch growing is
now just halted
      parentBranch.IsGrowing = false
   end
end
```

```
function Tree:SplitBranches(branch : table)
   if (self.growth_limit - self.growth_amount < 10) then
      return
   end

   local split_amount =
self:RandomIntFromTable(self.branch_split_amount)

   local branches = {}

   for _=1,200 do
      local can_grow = true

      -- needs to be #branches instead of 1 because it can *fail*!
      for _=1, split_amount do
         local y_angle = self:RandomFloatFromTable({minimum=0,
maximum=math.rad(360)})
         local angle = CFrame.Angles(0, y_angle,
math.rad(self:RandomFloatFromTable(self.branch_angle))) *
CFrame.Angles(0, -y_angle, 0)

         local branch_cframe = branch.root_location * CFrame.new(0,
branch.height, 0) * angle

         local look_vector = (branch_cframe *
CFrame.Angles(math.rad(90), 0, 0)).LookVector
         if (look_vector.Y > 1 or look_vector.Y < 0.3) then
            branches = {}
            can_grow = false
            break
         end

         -- thanks google
         local clamp = {
            minimum =
math.sqrt(math.cos(math.rad(self.angle_clamp.minimum)) * 2 - 4),
            maximum =
math.sqrt(math.cos(math.rad(self.angle_clamp.maximum)) * 2 - 4)
         }

         -- check distance
         for _,cframe in pairs(branches) do
            local magnitude = ((cframe).p -
(branch_cframe).p).magnitude

            if (magnitude > clamp.maximum or magnitude <
clamp.minimum) then
               branches = {}
               can_grow = false
               break
            end
         end

         table.insert(branches, branch_cframe)
      end

      if (can_grow) then break end
   end

   for _,location in pairs(branches) do
      self:CreateBranch(location, "Branch", branch.id)
   end

   if (#branches == 0) then
      branch.growth_enabled = false
   else
      self:DestroyLeaves(branch)
   end
end

. . .
```

```
. . .

function SuperTree:NewSection(cframe, originType, distanceUpParentBranch,
parentId, bendAngle)
   local parentSection = self.Sections[parentId]

   if originType == "Branch" then
      local branch = self.BranchClass[math.random(1, #self.BranchClass)]
      local newTree = require(branch.Class)
      local skipTrunkYield = branch.skipTrunkYield

      newTree = newTree.new(self.Model, self, parentSection)
      newTree.SkipTrunkYield = skipTrunkYield

      newTree:PlantSeed(cframe)

      local idValue = Instance.new("IntValue", newTree.Model)
      idValue.Name = "ParentTreeSectionId"
      idValue.Value = parentId
      local angleValue = Instance.new("CFrameValue", newTree.Model)
      angleValue.Name = "BranchAngle"
      angleValue.Value = bendAngle
      local distValue = Instance.new("NumberValue", newTree.Model)
      distValue.Name = "DistUpParentTreeSection"
      distValue.Value = distanceUpParentBranch

      table.insert(self.Branches, {Branch = newTree, ParentId =
parentId})
      return
   end

   local section = {}

   section.Part = newPart()
   section.Part.Name = "WoodSection"
   section.Part.BrickColor = self.BarkColor
   section.Part.Material = self.BarkMaterial
   section.Part.Transparency = self.BarkTransparency or 0
   section.Part.Reflectance = self.BarkReflectance or 0

   section.InsideWoods = {}


   if parentSection then --Came from another branch
      if originType == "Bend" then
         section.Thickness = parentSection.Thickness -
randomRange(self.BendThicknessReduce)
      elseif originType == "Split" then
         section.Thickness = parentSection.Thickness -
randomRange(self.SplitThicknessReduce)
      end
      section.AncestryLength = parentSection.Length +
parentSection.AncestryLength
   elseif originType == "Seed" then --Is a seed
      section.Thickness = randomRange(self.SeedThickness)
      section.AncestryLength = 0
   end
   section.Length = 0.2

   section.StartCFrame=cframe
   section.Part.Size = Vector3.new(section.Thickness, section.Length,
section.Thickness)
   section.Part.CFrame = section.StartCFrame * CFrame.new(0,
section.Length / 2, 0)
   section.Part.Parent = self.Model

   section.LeafBunch = self:NewLeafBunch(section)

   section.IsExtremety = true
   section.IsGrowing = true

   . . .

   section.ID = #self.Sections + 1
   section.ParentId = parentId

   local idValue = Instance.new("IntValue", section.Part)
   idValue.Name = "ID"
   idValue.Value=section.ID

   local parentValue = Instance.new("IntValue", section.Part)
   parentValue.Name = "ParentID"
   parentValue.Value = parentId

   local childrenValue = Instance.new("IntValue", section.Part)
   childrenValue.Name = "ChildIDs"

   if parentSection then
      table.insert(self.Sections[parentId].ChildrenSections, section.ID)

      local childValue = Instance.new("IntValue",
self.Sections[parentId].Part.ChildIDs)
```

```
. . .

function Tree:CreateBranch(location : CFrame, branch_type : string,
parent : number)
   local parent_branch = self.branches[parent]

   local branch = {}
   branch.children = {}
   branch.growth_enabled = true

   if (parent) then
      table.insert(self.branches[parent].children,#self.branches + 1)
   end

   branch.part = self:CreatePart("Branch")
   branch.part.Name = "VWOOD"

   branch.part.Color = self.bark_information.color
   branch.part.Material = self.bark_information.material
   branch.part.Transparency = self.bark_information.transparency

   -- Initial Sizing
   if (branch_type == "Root") then
      branch.thickness = self:RandomFloatFromTable(self.starting_size)
      branch.length_till_branch =
self:RandomFloatFromTable(self.root_branch_distance)
   else
      branch.thickness = self:RandomFloatFromTable(self.starting_size)
   end

   -- Descendant Sizing
   if (parent_branch and branch_type == "Branch") then
      branch.thickness = parent_branch.thickness -
self:RandomFloatFromTable(self.branch_size_loss)
      branch.length_till_branch =
self:RandomFloatFromTable(self.branch_distance)
   end

   -- Constraint Setup
   branch.height = self.starting_size.minimum
   branch.root_location = location

   -- Visual Setup
   branch.part.CFrame = branch.root_location * CFrame.new(0,
branch.height / 2, 0)
   branch.part.Size = Vector3.new(branch.thickness, branch.height,
branch.thickness)

   branch.part.Parent = self.model

   -- you have to create leaves after the part, that way they update
correctly
   branch.leaves = self:CreateLeaves(branch)

   -- Value Information
   branch.id = #self.branches + 1
   branch.parent = parent

   local parent_id_value = Instance.new("IntValue")
   parent_id_value.Parent = branch.part
   parent_id_value.Value = parent
   parent_id_value.Name = "ParentID"

   local id_value = Instance.new("IntValue")
   id_value.Parent = branch.part
   id_value.Value = branch.id
   id_value.Name = "ID"

   local child_value = Instance.new("IntValue")
   child_value.Name = "ChildIDs"
   child_value.Parent = branch.part


   if (parent_branch) then
      local child_value = Instance.new("IntValue")
      child_value.Name = "Child"
      child_value.Value = branch.id
      child_value.Parent = self.branches[parent].part.ChildIDs
   end

   self.branches[branch.id] = branch
end

. . .
```

```
      childValue.Name = "Child"
      childValue.Value = section.ID

      local angleValue = Instance.new("CFrameValue", childValue)
      angleValue.Name = "Angle"
      angleValue.Value = bendAnglet
   end
   section.ChildrenSections = {}

   self.Sections[#self.Sections + 1] = section
end

. . .

function SuperTree:NewLeafBunch(section)
   local numLeaves = randomRangeInteger(self.NumLeafParts)

   if game.Lighting.Spook.Value then
      numLeaves = 0
   end

   local leaves = {}
   for i = 1, numLeaves do --All the parts that make up a leaf bunch
      local newLeaf = {}

      newLeaf.LeafSizeFactor =
Vector3.new(randomRange(self.LeafSizeFactor.X),
                                 randomRange(self.LeafSizeFactor.Y),
                                   randomRange(self.LeafSizeFactor.Z))
      newLeaf.LeafAngle = CFrame.Angles(0, math.pi/2, 0) --Orient
correctly
                     * CFrame.Angles(0,
math.rad(randomRange(self.LeafAngle.Y)), 0) --Spin
                     *
CFrame.Angles(math.rad(randomRange(self.LeafAngle.X)), 0,
math.rad(randomRange(self.LeafAngle.Z))) --Tilt


      newLeaf.Part = newPart()
      newLeaf.Part.CanCollide = false
      local colorChoice =
self.LeafColors[math.random(1,#self.LeafColors)]
      newLeaf.Part.BrickColor = colorChoice.Color
      newLeaf.Part.Material = colorChoice.Material
      newLeaf.Part.Transparency = colorChoice.Transparency or 0
      newLeaf.Part.Reflectance = colorChoice.Reflectance or 0

      newLeaf.Part.Parent = self.LeafModel
      newLeaf.Part.Name = "LeafPart"

      leaves[i] = newLeaf
   end

   self:UpdateLeaves(section, leaves)

   return leaves
end

. . .

function SuperTree:UpdateLeaves(section, leavesIn)
   local leaves = leavesIn or section.LeafBunch
   for _, leaf in pairs(leaves) do
      if leaf.Part.Parent then
         leaf.Part.Size = leaf.LeafSizeFactor * section.Thickness
         leaf.Part.CFrame = section.StartCFrame * CFrame.new(0,
section.Length + 0.75 * leaf.Part.Size.Y/2, 0) * leaf.LeafAngle
      end
   end
end

. . .

function SuperTree:DestroyLeaves(section)
   local leaves = section.LeafBunch
   if leaves then
      for _, leaf in pairs(leaves) do
         if leaf.Part.Parent then
            leaf.Part:Destroy()
         end
      end
      self.Sections[section.ID].LeafBunch = {}
   end
end
```

```
function Tree:CreateLeaves(branch : table)
   local leaf_count = self:RandomIntFromTable(self.leaf_count)

   local leaves = {}

   for _=1,leaf_count do
      local new_leaf = {}

      new_leaf.size = Vector3.new(
         self:RandomFloatFromTable(self.leaf_size.x),
         self:RandomFloatFromTable(self.leaf_size.y),
         self:RandomFloatFromTable(self.leaf_size.z)
      )

      new_leaf.angle = CFrame.Angles(
         math.rad(self:RandomFloatFromTable(self.leaf_orientation.x)),
         math.rad(self:RandomFloatFromTable(self.leaf_orientation.y)),
         math.rad(self:RandomFloatFromTable(self.leaf_orientation.z))
      )

      local customization_properties =
self.leaf_information[self:RandomIntFromTable({
         minimum = 1,
         maximum = #self.leaf_information
      })]

      new_leaf.ignore_relativity =
customization_properties.ignore_relativity

      new_leaf.part = self:CreatePart("Leaf")
      new_leaf.part.CustomPhysicalProperties =
PhysicalProperties.new(0.1, 0, 0, 1, 1)
      new_leaf.part.CanCollide = false
      new_leaf.part.Massless = true

      new_leaf.part.Color = customization_properties.color
      new_leaf.part.Material = customization_properties.material
      new_leaf.part.Transparency =
customization_properties.transparency

      new_leaf.part.Parent = self.leaf_model
      new_leaf.part.Name = "Leaf"

      table.insert(leaves, new_leaf)
   end

   self:RefreshLeaves(branch, leaves)
   return leaves
end

. . .

function Tree:RefreshLeaves(branch : table, leaves : table)
   for _,leaf in pairs(leaves or branch.leaves) do
      if (leaf.part.Parent) then
         leaf.part.Size = leaf.size * branch.thickness * 1.5
         if (leaf.ignore_relativity) then
            leaf.part.CFrame = CFrame.new((branch.root_location *
CFrame.new(0, branch.height + 0.5 * leaf.part.Size.Y/2, 0) *
leaf.angle).p)-- * CFrame.new(0, leaf.part.Size.Y/2, 0)
         else
            leaf.part.CFrame = (branch.root_location * CFrame.new(0,
branch.height + 0.5 * leaf.part.Size.Y/2, 0) * leaf.angle)
         end
      end
   end
end

. . .

function Tree:DestroyLeaves(branch : table)
   if (branch.leaves) then
      for _,leaf in pairs(branch.leaves) do
         if (leaf.part.Parent) then
            leaf.part:Destroy()
         end
      end
      -- 10 points for memory leak
      branch.leaves = {}
   end
end
```